

华中科技大学

课程实验报告

课程名称： 计算机系统基础实验

专业班级： 大数据 2101 班

学 号： U202115652

姓 名： 李嘉鹏

指导教师： 李海波

实验时段： 2022 年 9 月 27 日~11 月 15 日

实验地点： 南一楼 808 室

原创性声明

本人郑重声明：本报告的内容由本人独立完成，有关观点、方法、数据和文献等的引用已经在文中指出。除文中已经注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品或成果，不存在剽窃、抄袭行为。

特此声明！

学生签名： 李嘉鹏

报告日期：2022 年 12 月 12 日

实验报告成绩评定：

	1	2	3	4	5	6
实验完成质量（70%），报告撰写质量（30%），每次满分 20 分。						
合计（100 分）						

备注：实验完成质量从实验目的达成程度，设计方案、实验方法步骤、实验记录与结果分析论述清楚等方面评价；报告撰写质量从撰写规范、完整、通顺、详实等方面评价。

指导教师签字：

日期：

汇编语言程序设计实验报告

目录

1	汇编语言编程基础.....	3
1.1	实验内容	3
1.2	任务 1.1 实验过程	3
1.2.1	实验方法说明	3
1.2.2	实验记录与分析	3
1.3	任务 1.2 实验过程	5
1.3.1	实验方法说明	5
1.3.2	实验记录与分析	6
1.4	任务 1.3 实验过程	7
1.4.1	实验方法说明	7
1.4.2	实验记录与分析	7
1.5	任务 1.4 的实验过程	9
1.5.1	设计思想及存储单元分配	9
1.5.2	流程图	9
1.5.3	实验步骤说明	10
1.5.4	实验记录与分析	10
1.6	小结	11
2	程序优化.....	13
2.1	实验内容	13
2.2	任务 2 实验过程	13
2.2.1	实验方法说明	13
2.2.2	实验记录与分析	13
2.2.3	用 C 语言模拟并观察 debug 和 release 两种模式的区别	14
2.3	小结	15
3	模块化程序设计.....	16
3.1	实验内容	16
3.2	任务 3.1 实验过程	16
3.2.1	设计思想及存储单元分配	16
3.2.2	流程图	17
3.2.3	实验步骤说明	17
3.2.4	实验记录与分析	18

汇编语言程序设计实验报告

3.3	任务 3.2 实验过程	22
3.3.1	实验方法说明	22
3.3.2	实验记录与分析	22
3.4	小结	24
4	二进制炸弹破解	25
4.1	实验内容	25
4.2	任务 4.1 实验过程	25
4.2.1	实验方法说明	25
4.2.2	实验记录与分析	25
4.3	小结	27
5	不同指令集体系结构和编程环境的比较	28
5.1	实验内容	28
5.2	任务 5.1 实验过程	28
5.2.1	实验方法说明	28
5.2.2	实验记录与分析	28
5.2.3	查阅资料并比较 ARMv8 与 80X86 体系的异同	30
5.3	小结	31
6	中断处理	32
6.1	实验内容	32
6.2	任务 6.1 实验过程	32
6.2.1	实验方法说明	32
6.2.2	设计思想	32
6.2.3	流程图	32
6.2.4	实验记录与分析	33
6.3	小结	36
	参考文献	37

1 汇编语言编程基础

1.1 实验内容

本次实验的主要目的与要求有：

- (1) 熟练掌握 DOSBox 下 16 位汇编语言程序开发工具的基本用法，包括程序的编译、链接和调试；
- (2) 熟悉编程的基础知识，包括数据在计算机内的表现形式、寻址方式、常用指令等；
- (3) 熟悉程序运行的基本原理；
- (4) 熟悉分支、循环程序的结构及控制方法，掌握分支、循环程序的调试方法；
- (5) 加深对转移指令及一些常用汇编指令的理解。

1.2 任务 1.1 实验过程

1.2.1 实验方法说明

1. 准备上机实验环境，对实验用到的软件进行安装、运行，通过试用初步了解软件的基本功能、操作等。

2. 在 TD 的代码窗口中的当前光标下输入第一个运算式对应的两个 8 位数值对应的指令语句，即：MOV AH,00110011B; MOV AL,01011010B; ADD AH,AL; 观察代码区显示的内容与输入字符之间的关系；然后确定 CS:IP 指向的是自己输入的第一条指令的位置，单步执行三次，观察寄存器内容的变化，记录标志寄存器的结果。

3. 对课本习题中给出的另外两组操作数，重复上述流程。分析程序执行的结果，并回答任务书中相应问题。

1.2.2 实验记录与分析

1. 实验环境条件：P3 1GHz，256M 内存；WINDOWS 10 下 DOSBox0.73；TD.EXE 5.0。

2. (1) 第一组指令执行过程与结果分析

在 TD 中直接输入第一组指令，共 3 条：MOV AH,00110011B; MOV AL,01011010B; ADD AH,AL。此时 CS:IP 指向输入的第一条指令。

依次执行这三条指令并观察代码区内容发生的变化，如图 1.1 所示。此时标志位 CF（进位标志）、ZF（零标志）、SF（符号标志）、OF（溢出标志）的状态应该分别为 SF=1，OF=1，CF=0，ZF=0，其理由在于：00110011 和 01011010 做二进制加法时，得到的结果为 10001101（十六进制下表示为 8DH，即图 1.1 中 AX 寄存器值的高 2 位）。此结果不等于 0，故 ZF=0。若将其视为有符号数，则超过了 7 位二进制数（因为最高位为符号位）的表示范围，因此 OF 等于 1；且此结果为负，SF 等于 1。但若将这两个二进制数看做无符号数，则结果没有超过 8 位二进制数的表示范围，也没有发生进位，因此 CF 为 0。上述分析与预估结果一致，且（AH）的值为 8DH。

图 1.1 执行完第一组测试语句（求和）后的状态

图 1.2 执行完第一组测试语句（求差）后的状态

图 1.3 第一组测试语句求和后的状态 (若不输入 B)

表 1.1 对后两组语句执行结果的预测

相应结果如图 1.4-1.7 所示。

汇编语言程序设计实验报告

cs:0100 B4A9	mov	ah,A9		
cs:0102 B0DD	mov	al,DD		
cs:0104 02E0	add	ah,al		
cs:0106 1E	push	ds		
cs:0107 8ED8	mov	ds,ax		
cs:0109 8E4608	mov	es,[bp+08]		
cs:010C 8CC3	mov	bx,es		
cs:010E B80600	mov	ax,0006		
			bx 0000	z=0
			cx 0000	
			dx 0000	o=0
			si 0000	p=0
			di 0000	
			bp 0000	i=1
			sp 0080	d=0

图 1.4 执行完第二组测试语句（求和）后的状态

cs:0100 B4A9	mov	ah,A9		
cs:0102 B0DD	mov	al,DD		
cs:0104 2AE0	sub	ah,al		
cs:0106 1E	push	ds		
cs:0107 8ED8	mov	ds,ax		
cs:0109 8E4608	mov	es,[bp+08]		
cs:010C 8CC3	mov	bx,es		
cs:010E B80600	mov	ax,0006		
			bx 0000	z=0
			cx 0000	
			dx 0000	o=0
			si 0000	
			di 0000	
			bp 0000	i=1
			sp 0080	d=0

图 1.5 执行完第二组测试语句（求差）后的状态

cs:0100 B4A9	mov	ah,A9		
cs:0102 B0DD	mov	al,DD		
cs:0104 2AE0	sub	ah,al		
cs:0106 B465	mov	ah,65		
cs:0108 B0DD	mov	al,DD		
cs:010A 02E0	add	ah,al		
cs:010C 8CC3	mov	bx,es		
cs:010E B80600	mov	ax,0006		
			bx 0000	c=1
			cx 0000	z=0
			dx 0000	o=0
			si 0000	p=1
			di 0000	a=1
			bp 0000	i=1
			sp 0080	d=0

图 1.6 执行完第三组测试语句（求和）后的状态

cs:0100 B4A9	mov	ah,A9		
cs:0102 B0DD	mov	al,DD		
cs:0104 2AE0	sub	ah,al		
cs:0106 B465	mov	ah,65		
cs:0108 B0DD	mov	al,DD		
cs:010A 02E0	add	ah,al		
cs:010C B465	mov	ah,65		
cs:010E B0DD	mov	al,DD		
cs:0110 2AE0	sub	ah,al		
cs:0112 317216	xor	[bp+si+16],si		
			bx 0000	c=1
			cx 0000	z=0
			dx 0000	
			si 0000	p=1
			di 0000	a=1
			bp 0000	i=1
			sp 0080	d=0
			ds 0A96	
			es 0A96	

图 1.7 执行完第三组测试语句（求差）后的状态

（3）对任务书上问题的回答

根据以上结果可得：求差运算中，若将两个操作数 A、B 视为有符号数，且 $A > B$ ，则最终标志位的特点是 $SF=0$ ， $OF=0$ ， $ZF=0$ （结果不可能等于零）， CF 可能为 0 或 1 但无意义。

若将 A、B 视为无符号数，且 $A > B$ ，则标志位的特点是 $OF=0$ 但无意义， CF 可能为 0 或 1， $ZF=0$ ， SF 为结果的最高位（但实际上无意义）。

1.3 任务 1.2 实验过程

1.3.1 实验方法说明

1. 通过记事本完成对汇编代码的创建和写入，然后通过 DOSBox 将源程序文件.asm 先后通过 masm 和 link 指令转换为可执行目标文件.exe，然后在 TD 中运行、调试。分别记录运行到两个关键位置“MOV CX, 10”和“INT 21H”时寄存器 BX、BP、SI、DI 存储的数据。

2. 记录数据段开始 40 个字节的内容，并判断是否与预期结果一致。

汇编语言程序设计实验报告

3. 在标号 LOPA 前加入一段程序，并实现先给出提示信息后通过按任意键继续的功能。

1.3.2 实验记录与分析

(1) 汇编代码的编写和可执行目标文件的创建

首先，在记事本中输入课本提供的完整程序代码。然后在 DOSBox 中先后输入 C:\>MASM 1-2, C:\>LINK 1-2, C:\>TD 1-2, 如图 1.8 所示。MASM 指令可以汇编源文件并生成目标文件.obj, LINK 指令可以连接目标文件并生成可执行文件.exe。

```
C:\>MASM 1-2
Microsoft (R) MASM Compatibility Driver
Copyright (C) Microsoft Corp 1991. All rights reserved.

Invoking: ML.EXE /I. /Zm /c /Ta 1-2.asm

Microsoft (R) Macro Assembler Version 6.00
Copyright (C) Microsoft Corp 1981-1991. All rights reserved.

Assembling: 1-2.asm

C:\>LINK 1-2

Microsoft (R) Segmented Executable Linker Version 5.20.034 May 24 1991
Copyright (C) Microsoft Corp 1984-1991. All rights reserved.

Run File [1-2.exe]:
List File [nul.map]:
Libraries [.lib]:
Definitions File [nul.def]:
```

图 1.8 对汇编代码文件进行汇编和连接

(2) 程序执行过程的观察

程序执行到“MOV CX, 10”时已经完成了对一些基本寄存器的赋值操作，记录此时的 (BX)=0014H, (BP)=001EH, (SI)=0000, (DI)=000A, 它们分别代表 BUF1、BUF2、BUF3、BUF4 四个存储区的首偏移地址，如图 1.9 所示。

cs:0000 B8BD0A	mov	ax,0ABD	ax 0ABD	c=0
cs:0003 8ED8	mov	ds,ax	bx 0014	z=0
cs:0005 BE0000	mov	si,0000	cx 0000	s=0
cs:0008 BF0A00	mov	di,000A	dx 0000	o=0
cs:000B BB1400	mov	bx,0014	si 0000	p=0
cs:000E BD1E00	mov	bp,001E	di 000A	a=0
cs:0011 B90A00	mov	cx,000A		i=1

图 1.9 第一次记录四个寄存器的值

在“INT 21H”之前设置断点，并直接执行到断点处，再次记录此时的 (BX)=001EH, (BP)=0028H, (SI)=000A, (DI)=0014, 如图 1.10 所示。可以发现这四个值相对初始值都增加了 10，代表访问数据存储单元过程中走过的个数。

此时在下方内存区域右键 Goto 输入“DS: 00”可以查看本程序占用的 40 个字节空间，其值如图 1.10 所示。可以看到前 10 个字节和第 11~20 个字节的值均为 00~09，第 21~30 个字节的值为 01~0A（由 00~09 分别加 1 得到），第 31~40 个字节的值为 04~0D（由 00~09 分别加 4 得到）。而根据程序功能，每次循环都会访问 BUF1、BUF2、BUF3、BUF4 四个存储区的下一元素，其中 BUF2 复制 BUF1 元素的值，BUF3 在 BUF1 元素值的基础上加 1，BUF4 在 BUF3 的基础上加 3，因此运行结果与设想情况一致。

汇编语言程序设计实验报告

接下来，在标号 LOPA 前加入一段程序。首先在数据段中加入要输出的字符串：DATA SEGMENT USE16 BUF5 DB 'Press any key to begin!\$', 然后在 LOPA 前加入 LEA DX, BUF5, MOV AH, 9, INT 21H 三条语句，将当前数据区中 DX:DX 所指的以“\$”结尾的字符串在显示器上显示。然后还要在 LOPA 前加入 MOV AH, 1, INT 21H 从而实现等待用户输入一个字符的效果，如图 1.11 所示。

cs:0029 B44C	mov	ah,4C		c=0
cs:002B CD21	int	21	bx 001E	z=1
cs:002D B226	mov	d1,26	cx 0000	s=0
cs:002F 3B06E700	cmp	ax,[00E7]	dx 0000	o=0
cs:0033 7402	je	0037	si 000A	p=1
cs:0035 EB2E	jmp	0065	di 0014	a=0
cs:0037 0306B426	add	ax,[26B4]	bp 0028	i=1
cs:003B 0306B626	add	ax,[26B6]	sp 00C8	d=0
cs:003F A3B026	mov	[26B0],ax	ds 0ABD	
cs:0042 BB0500	mov	bx,0005	es 0AA0	
cs:0045 F7E3	mul	bx	ss 0AB0	
cs:0047 0BD2	or	dx,dx	cs 0AC0	
cs:0049 7402	je	004D		
cs:004B EB12	jmp	005F		
cs:004D A3BC26	mov	[26BC],ax		

ds:0000 00 01 02 03 04 05 06 07	00 01 02 03 04 05 06 07	ss:00CA 0000
ds:0008 08 09 0A 0B 0C 0D 0E 0F	00 01 02 03 04 05 06 07	ss:00CB 0000
ds:0010 06 07 08 09 0A 0B 0C 0D	00 01 02 03 04 05 06 07	ss:00CC 0000
ds:0018 05 06 07 08 09 0A 0B 0C	00 01 02 03 04 05 06 07	ss:00CD 07FA
ds:0020 06 07 08 09 0A 0B 0C 0D	00 01 02 03 04 05 06 07	ss:00CE 0A95

图 1.10 第二次记录四个寄存器的值并观察数据段开始 40 个字节的内容

```
C:\>td 1-2
Turbo Debugger Version 5.0 Copyright (c) 1988,96 Borland International
Press any key to begin!
```

图 1.11 实现新的功能，等待用户按一个键后继续执行

1.4 任务 1.3 实验过程

1.4.1 实验方法说明

1. 根据要求，将访问数据段变量的寻址方式中的寄存器改为 32 位寄存器，并将对内存操作数的访问改为变址寻址方式。记录程序退出之前数据段开始 40 个字节的内容，并判断是否与预期结果一致。
2. 在 TD 中观察机器指令在内存中的存放形式，重点理解操作数寻址的指令和反汇编代码之间的区别，并与任务 1.2 作对比。
3. 观察连续存放的二进制串在反汇编时从不同位置开始的影响，理解 IP/EIP 指明指令起始位置的重要性。

1.4.2 实验记录与分析

(1) 对代码的修改

首先将任务 1.2 中汇编代码进行修改，主要包括以下两点：第一，将访问数据段中所用寻址方式中的寄存器改成 32 位寄存器，代码改为 MOV EAX, DATA, MOV DS, EAX, MOV ESI, 0，其中 EAX、ESI 为 32 位寄存器；第二，采用变址寻址方式访问内存单元中的数据，即将 ESI 初值赋

汇编语言程序设计实验报告

为 0（代表 BUF1~BUF4 的“变址”，会不断改变），每循环一次就加 1，寻址时的格式为[寄存器+偏移地址]，即 MOV AL, [ESI+BUF1], MOV [ESI+BUF2], AL, MOV [ESI+BUF3], AL, MOV DS:[ESI+BUF4], AL。

执行后，记录程序退出之前数据段开始 40 个字节的内容，如图 1.12 所示，结果与预期一致。

```
ds:0000 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
ds:0008 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27
ds:0010 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27
ds:0018 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27
ds:0020 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27
```

图 1.12 修改代码后观察数据段开始 40 个字节的内容

对于本题，一开始我考虑引入一个 COUNT 变量记录循环次数，并通过 MOV AL,[ESI+COUNT] 的方式实现变址寻址，但发现结果不对，原因在于[COUNT]位于中括号内部，程序反汇编时会取 COUNT 的偏移地址，而不是其内部存储的值，显然会导致目标操作数位置错误。如图 1.13 所示，反汇编代码中[ESI+COUNT]被转化为[ESI+00000040]，因为 COUNT 的偏移地址是 40H。进一步我发现可以只通过 ESI 寄存器存储变址。

```
cs:002B 678A864000000000 mov al,[esi+00000040]
cs:0032 6788874000000000 mov [edi+00000040],al
cs:0039 FEC0 inc al
cs:003B 6788834000000000 mov [ebx+00000040],al
```

图 1.13 不能直接使用变量进行变址寻址

（2）观察并比较汇编代码与反汇编代码的区别

程序中共有几种不同类型的语句。在 TD 中，可以看到全部指令的反汇编代码，其中每一条最左边是指令相对 CS 段首址的偏移地址，最右边是反汇编代码，中间是该指令的机器码。

```
cs:0000 66B8BD0A0000 mov eax,0000ABD
cs:0006 66BED8 mov ds,ax
cs:0009 66BE00000000 mov esi,00000000
cs:000F B90A00 mov cx,000A
cs:0012 BA2800 mov dx,0028
cs:0015 B409 mov ah,09
cs:0017 CD21 int 21
cs:0019 678A860000000000 mov al,[esi]
cs:0020 6788860A00000000 mov [esi+0000000A],al
cs:0027 FEC0 inc al
cs:0029 6788861400000000 mov [esi+00000014],al
cs:0030 0403 add al,03
cs:0032 6788861E00000000 mov [esi+0000001E],al
cs:0039 6646 inc esi
cs:003B 49 dec cx
cs:003C 75DB jne 0019 ↑
```

图 1.14 在 TD 中观察反汇编代码

有的反汇编代码与汇编代码没有区别，例如 INC AL 的反汇编代码为 inc al；有的反汇编代码直接将偏移地址显化，例如 LEA DX, BUF5 的反汇编代码为 mov dx, 0028（其中 0028 代表 BUF5 的偏移地址，在运行时确定），变址寻址的反汇编代码也满足这一特点，如 MOV [ESI+BUF2], AL 的反汇编代码为 mov [esi+0000000A], al（ESI 是寄存器，不会直接转换为地址，但 BUF2 的偏移地址被显化）。同样地，程序段中的标号（如 START、LOPA）也会直接转化为地址（在 jump 指令中可以观察到）。而在任务 1.2 中，MOV AL, [SI] 的反汇编代码就是 mov al, [si]，可见寄存器名在反汇

汇编语言程序设计实验报告

编时会维持原状。还要注意的，反汇编代码中常数是用 16 进制存储的。以上各类语句如图 1.14 所示。

(3) 观察从不同位置开始反汇编的程序运行状态

在 TD 代码区右键 Goto 输入 “CS:1000” (不在任务汇编代码范围内) 并开始运行，发现此时 ip=1000，CS:1000 位置的代码行出现白色右箭头，说明程序目前运行到该语句。可见，IP/EIP 寄存器存储的是当前语句的地址，对程序正常运行有很大意义。如果不慎错误修改了 IP/EIP 的值，则会产生不可预见的后果。

1.5 任务 1.4 的实验过程

1.5.1 设计思想及存储单元分配

本题要实现一个数据处理程序。根据题目要求，直接对 a、b、c 三个变量进行运算并设置合理的跳转条件，即可将各组数据依据计算结果进行分组复制。

1. 存储单元分配

SAMID: 字节变量，存放每组数据的流水号。

SDA、SDB、SDC: 双字变量，分别存放 10 组数据的三个状态信息。

SF: 双字变量，存储每次处理的结果。

LOWF、MIDF、HIGHF: 双字变量，用于复制并存储三种数据。

COUNT: 字节变量，本质是计数器，初值为 10，代表循环 10 次。

RECORD: 双字变量，用于变址寻址中过渡。

2. 寄存器分配

ESI: 存放循环次数，利用变址寻址方式访问某一组数据的状态信息。

EBP、EDI: 存储两个复制数据段的当前变址。

EAX、EBX、ECX、DH: 临时寄存器。

1.5.2 流程图

图 1.15 是任务 1.4 数据处理程序的程序流程图。

汇编语言程序设计实验报告

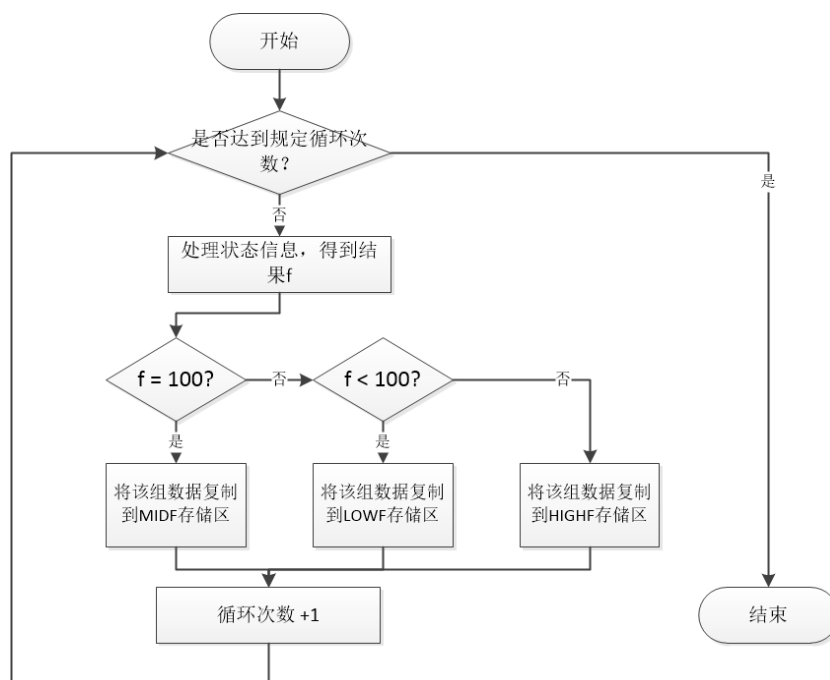


图 1.15 数据处理程序的程序流程图

1.5.3 实验步骤说明

1. 准备上机实验环境。
2. 使用记事本录入源程序，存盘文件名为 1-4-original.ASM。使用 MASM 6.0 汇编源文件。即 MASM 1-4-original；观察提示信息，若出错，则用编辑程序修改错误，存盘后重新汇编，直至不再报错为止。
3. 使用连接程序 LINK.EXE 将汇编生成的 1-4-original.OBJ 文件连接成执行文件。即 LINK 1-4-original；若连接时报错，则依照错误信息修改源程序。之后重新汇编和连接，直至不再报错并生成 1-4-original.EXE 文件。
4. 执行该程序。即在命令行提示符后输入 1-4-original 后回车，观察执行现象和结果。
5. 使用 TD.EXE 观察程序的执行情况。即 TD 1-4-original.EXE 回车
 - (1) 观察 CS、IP、SP、DS、ES、SS 的值。
 - (2) 单步执行开始 2 条指令，观察 DS 的改变情况。
 - (3) 观察 SS: 0 至 SS: SP 区域的数据值。
 - (4) 通过 Goto 指令观察 DS: 0 开始的数据区，找到各变量在数据段中的位置和值。
 - (5) 观察指令的表现形式，与上面分析的 TD 反汇编与汇编代码之间的联系进行对照。
 - (6) 按住 F7 循环执行程序，观察 ESI 的变化情况；ESI 变为 000A 后，在数据段中观察三个存储区的值，检验复制功能是否实现。

1.5.4 实验记录与分析

1. 实验环境条件：P3 1GHz，256M 内存；WINDOWS10 下 DOSBox0.73；EDIT.EXE 2.0；MASM.EXE 6.0；LINK.EXE 5.2；TD.EXE 5.0。
2. 汇编源程序时，汇编程序报了 1 个错误：

汇编语言程序设计实验报告

1-4-original.asm(22): error A2022: instruction operands must be the same size

按照提示，对对应行仔细检查，发现了错误原因，见源程序中“;”后的说明。

3. 连接过程没有发生异常。

4. 用 TD 调入 1-4-original.EXE 后：

(1) (CS)=0DB4H、(IP)=0000H、(SP)=00C8H、(DS)=0AA0H、(ES)=0AA0H、(SS)=0AB0H。

可从段首址取值的不同得知代码段、数据段、堆栈段处在不同位置，且前后次序是数据段、堆栈段、代码段。

(2) 单步执行开始 2 条指令，(DS)→0ABDH。观察到 TD 的数据显示区被切换到了 ES:0，若还希望显示 DS:0，就需要用 Goto 指令重新设定。可以发现此时数据段才是程序的实际位置，各个段的实际次序为堆栈段、数据段、代码段，这与源程序定义各段的次序一致。

(3) SS: 0 至 SS: SP 区域的数据值在程序没有执行时均为 0。

(4) DS: 0 开始数据区存放了数据段中定义的各类数据，以 SAMID 数据段为首地址开始存储。EA=0AH 开始存放 SDA，EA=32H 开始存放 SDB，EA=5AH 开始存放 SDC，EA=82H 存放 SF（当前值为 0）。此后，EA=86H、EA=46EH、EA=856H 分别存放 LOWF、MIDF、HIGHF（当前值为 0）；EA=0C3EH 存放 COUNT（当前值为 10），EA=0C3FH 存放 RECORD（当前值为 0）。

(5) TD 中显示的第三条语句为 MOV ESI, 00000000，可见是按照双字处理的；第四条语句为 MOV DWORD PTR [2F67], 00000000，说明在数据段中定义的变量是内存操作数，直接通过地址来访问其内容。

(6) 按住 F7，程序依次处理 10 组数据，观察到 ESI 从 0000 开始不断增长最终变为 000A。依次在数据段中查看 LOWF、MIDF、HIGHF 的内容，与预期结果一致，如图 1.16 所示。

```
ds:0086 01 01 00 00 00 01 00 00 00 00 00 00 00 00 00
ds:008E 00 01 00 00 00 02 02 00 00 00 00 00 00 00 00
ds:0096 00 00 02 00 00 00 02 00 00 00 00 00 00 00 00
ds:009E 00 00 03 03 00 00 00 03 00 00 00 00 00 00 00
ds:00A6 00 00 00 03 00 00 00 04 00 00 00 00 00 00 00
ds:046E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ds:0476 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ds:047E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ds:0486 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ds:048E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ds:0856 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ds:085E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ds:0866 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ds:086E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ds:0876 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

图 1.16 程序结束前三个存储区的状态

1.6 小结

通过本次实验，我掌握了利用 DOSBox 和 TurboDebugger 对汇编语言源程序进行编写、调试和观察的方法，以及汇编程序的编译、链接等环节的作用。在实验中，我深入了解了在汇编程序中标志位、寄存器各自承担的作用以及它们内部的细节与差异，熟悉了汇编语言中一些常用指令的用法与常见错误，对汇编与反汇编代码之间的联系有了更深的理解。我圆满完成了实验任务，对任务书所给的要求与提问均做了实现和解答。

汇 编 语 言 程 序 设 计 实 验 报 告

对于任务书最后给出的思考题（任务 1.4 中，如果三个状态信息是无符号数，程序需要做什么调整），我认为需要在改变除法的类型，即将有符号数的除法 IDIV 改为无符号数的除法 DIV，同时也不能使用 CDQ 进行被除数的符号扩展，否则会导致结果错误。

在实验过程中，我曾遇到过不少问题。其一是我最开始想类比 C 语言中访问数组的方式，利用一个常量实现变址寻址，但在 TD 中执行时发现翻译出的反汇编代码与真正想表达的意思不同，汇编器会取变量的地址而不是变量本身的值来进行寻址。于是我考虑使用一个寄存器对变量中所存的值进行过渡，就可以正常使用了。其二是在给寄存器赋值时，没有直接声明内存操作数的类型，导致汇编时报错提示两个操作数的容量大小不相等。其三是在 TD 中直接输入指令时，我起初不清楚代码段中为什么会出现大片无意义代码，后来经老师解答得知是 TD 将内存中某一段的当前值强行翻译为指令，这进一步提示我机器指令本质上就是存储在内存中的数据。

2 程序优化

2.1 实验内容

本次实验的主要内容是：

- (1) 了解程序计时的方法以及运行环境对程序执行情况的影响；
- (2) 深刻理解 CPU 执行指令的过程，不同特点的编程技巧和指令序列组合对程序长度及执行效率的影响，掌握代码优化的基本方法。

2.2 任务 2 实验过程

2.2.1 实验方法说明

1. 准备上机实验环境，对实验用到的软件进行安装、运行，初步了解软件的基本功能、操作等。
2. 将实验一任务 1.4 中的汇编代码进行改写，使其能在 VS2022 中运行（除此之外不做其它修改）。对该程序重复执行一定次数，明确计时方法并记录执行时间。
3. 对上述程序进行指令级别的优化，从不同角度提高源程序的性能。在同样的条件下运行优化后的程序，记录执行时间。分析比较优化效果，并尝试解释其原因。

2.2.2 实验记录与分析

1. 实验环境条件：P3 1GHz，256M 内存；WINDOWS 10 下 Visual Studio 2022。
2. (1) 优化前程序运行结果分析

为了使汇编程序能在 VS 中正常运行，首先要对其作出一定调整。为此需要在程序开头使用“.686P”来启用汇编指令并引入一些库。同时，由于本程序中需要用到 C 语言中的 clock 和 printf 函数，因此要使用 proto c 进行函数原型声明。

由于汇编代码效率高，单次执行几乎看不出执行时间，因此在程序主体外部再添加一层循环，使程序重复执行 5000000 次，即可使时间变得明显。在如此大的执行次数下，时间的统计只需要精确到毫秒级别就能满足要求。因此采用 clock 函数，在程序执行前获取一个时间标签，执行后再获取一个时间标签，然后通过计算这两个时间标签的差值来获取程序执行时间，可得执行时间为 7134 毫秒，如图 2.1 所示。经多次重复，发现每次执行的时间不完全一样但浮动不大，推测是由于系统硬件或进程等多种因素影响。

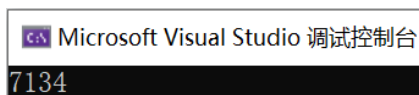


图 2.1 优化前程序执行时间

(2) 优化后程序运行结果分析

仔细考察原程序，发现其中有多处可以优化。第一处是对于有符号数的除法，若使用 IDIV 指

汇编语言程序设计实验报告

令，则涉及到一系列的内部操作，需要较多时间。注意到题目所给的除数为 128，恰好为 2 的 7 次方，于是可以联想到非循环算术右移位指令 SAR 可以方便地将一个数除以 2 的整数次幂。因此，可将原程序中“CDQ”“MOV ECX, 128”“IDIV ECX”简单地改成“SAR EAX, 7”，这样不仅不需要使用 IDIV 指令，也减少了两条指令。改进后在同等条件下测得执行时间为 3470 毫秒，如图 2.2 所示。可见优化的效果很好，达到了超过 50% 的优化率。

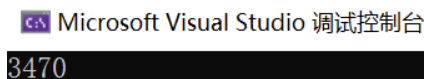


图 2.2 优化 IDIV 指令后程序执行时间

除了这一点外，在 CMP 指令后的分支中，对第三个判断条件可以去掉跳转指令，程序也会自动向下执行。这样优化可以减少 $5000000 \times 1 = 5000000$ 条指令的执行时间，当然，这样做也会降低程序的可读性。改进后在同等条件下测得执行时间为 2935 毫秒，如图 2.3 所示。

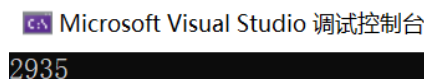


图 2.3 优化分支结构后程序执行时间

此外，我还考虑利用机器指令的特点对程序进行优化。但由于源程序本身就采用了基址+变址的寻址方式，因此此方面基本达到了最佳状态。

综上所述，总体优化率约为 $(7134 - 2935) / 7134 = 58.9\%$ 。

2.2.3 用 C 语言模拟并观察 debug 和 release 两种模式的区别

使用 C 语言模拟实现任务 1.4，并观察在 debug 和 release 两种模式下生成的执行程序的区别。主要区别在于：

(1) release 模式下采用寄存器 ECX 保存循环次数，而 debug 模式下创建了内存操作数 i，每次对 i 的操作需要用寄存器进行过渡，语句数量更多，如图 2.4 所示。

```
for (int i = 0; i < 10; i++)
00007FF7F40A171F  xor          ecx, ecx
00007FF7F40A1723  cmp          ecx, 0Ah
00007FF7F40A1723  jnl          ___$EncStackInitStart+2F4h (07FF7F40A1873h)
```

图 2.4 对循环次数 i 的寄存器优化

(2) release 模式下除 128 的操作优化为了 SAR 指令，而 debug 模式下还是按照算式中定义的顺序进行计算，更加繁琐，如图 2.5 所示。

(3) 程序段中涉及到多次对寄存器赋值为 0 的操作。release 模式下优化为指令“xor eax, eax”，通过位操作来给寄存器赋 0，而 debug 模式下只是简单地用 mov 指令来完成，推测是位操作指令比数据传送指令速度更快，如图 2.6 所示。

(4) 循环次数较少时（例如此处循环次数为 10），循环在 release 模式下被优化为顺序结构，即顺序执行 10 组数据的处理过程，避免了对循环次数的维护。

可见 release 模式从多个角度对源程序进行了优化。

汇编语言程序设计实验报告

```
f = (5 * a[i] + b[i] - c[i] + 100) / 128;
00007FF7F40A198B movsxd    rax,dword ptr [rbp+3C4h]
00007FF7F40A1992 imul     eax,dword ptr a[rax*4],5
00007FF7F40A1997 movsxd    rcx,dword ptr [rbp+3C4h]
00007FF7F40A199E add      eax,dword ptr b[rcx*4]
00007FF7F40A19A5 movsxd    rcx,dword ptr [rbp+3C4h]
00007FF7F40A19AC sub      eax,dword ptr c[rcx*4]
00007FF7F40A19B3 add      eax,64h
00007FF7F40A19B6 cdq      ►
00007FF7F40A19B7 and      edx,7Fh
00007FF7F40A19BA add      eax,edx
00007FF7F40A19BC sar      eax,7
00007FF7F40A19BF mov      dword ptr [f],eax
```

图 2.5 对除法运算的优化

```
return 0;
00007FF7F40A1BCB xor      eax,eax
```

图 2.6 对赋 0 操作的优化

2.3 小结

本次实验主要是研究怎样优化一个程序，使其达到更佳的性能。在实验中，我学会了利用 C 语言函数 `clock` 统计程序运行时间从而检测程序效率的方法。同时，我也对汇编指令的灵活运用与优化有了更多的认识，例如利用非循环算术移位指令代替有符号除指令（优化原理在于有/无符号除指令 `IDIV/DIV` 涉及较繁琐的内部运算，而算术移位指令 `SAR/SHR` 实现起来要简单的多），又比如在分支结构中删去最后一个条件的跳转指令（巧妙运用汇编语言的特性，因为指令是顺序执行的）。这提示我在今后的程序设计任务尤其是大型复杂计算程序中，需要格外关注底层指令的优化，仅仅是些微的差别就可能会导致程序的性能产生极大变化。

通过不断的探索，我最终对源程序实现了约 60% 的优化率，完成了实验要求。

3 模块化程序设计

3.1 实验内容

本次实验的主要目的与要求有：

- (1) 掌握子程序设计的方法与技巧，熟悉子程序的参数传递方法和调用原理；
- (2) 掌握模块化程序的设计方法；
- (3) 掌握汇编语言程序与 C 语言程序混合编程的方法；
- (4) 掌握较大规模程序的开发与调试方法，理解模块之间的信息传递与组装的基本方法；
- (5) 完成指定功能的程序设计与调试。

3.2 任务 3.1 实验过程

3.2.1 设计思想及存储单元分配

本题要在任务 1.4 的基础上实现一个计算机系统运行状态的监测系统。这里与此前处理数据程序的不同之处在于其涉及到了用户的输入输出，可能会根据用户的输入而退出程序或将程序重复执行多次，体现了分支与循环结构以及汇编语言中系统功能调用的相关内容。

1. 存储单元分配

SAMID：字节变量，存放每组数据的流水号。

SDA、SDB、SDC：双字变量，分别存放 10 组数据的三个状态信息。

SF：双字变量，存储每次处理的结果。

LOWF、MIDF、HIGHF：双字变量，用于复制并存储三种数据。

POS、POS1、POS2：双字变量，用于记录复制过程中三组数据的当前位置（地址）。

HINT、HINT1、HINT2、HINT3、SHOWHINT、SHOWHINT1、CHOICEHINT：字节变量，存储提示信息。

USERNAME、PASSWORD：存储正确的用户名和密码。

USERNAMEREAD、PASSWORDREAD：存储用户输入的用户名和密码。

NEWLINE：字节变量，其值为换行符。

PARTITION：字节变量，其值为一个逗号，用于在输出时分隔 MIDF 存储区的数据。

TEMP：字节变量，用于在复制过程中暂存流水号（与流水号类型对应）。

TEMP1：双字变量，用于在复制过程中暂存状态信息 a、b、c（同样与其类型对应）。

CHOICE：字节变量，存储字符‘R’，用于比对用户输入。

CHOICEREAD：存储用户输入的字符。

lpfmt、lpfmt1、lpfmt2、lpfmt3、lpfmt4：字节变量，存储 printf 函数的输出格式。

t：局部字变量，在 login 子程序中记录用户登录尝试次数，超过 3 则退出程序。

2. 寄存器分配

ESI：存放循环次数，利用变址寻址方式访问某一组数据的状态信息。

汇编语言程序设计实验报告

EDI: 在 getf 子程序中, 存放当前正在处理的数据元素下标; 在 showmid 子程序中, 存储 MIDF 最后一个数据的偏移地址。

EBP、EDI: 存储两个复制数据段的当前变址。

ECX: 在 login 子程序中标记用户登录是否成功, 是子程序的出口参数。

EAX: 在 login 子程序中记录用户输入的用户名和密码; 在 finalchoice 子程序中, 记录用户输入与字符 ‘R’ 用 strcmp 函数比较的结果。

其它寄存器: 临时寄存器。

EBX: 在 showmid 子程序中记录 MIDF 存储区当前的偏移地址; 在 finalchoice 子程序中标记用户输入是否为字符 ‘R’, 为子程序的出口参数。

3.2.2 流程图

图 3.1 是任务 3.1 计算机系统运行状态监测系统的程序流程图, 分为用户登录、数据处理、程序分支三个模块。

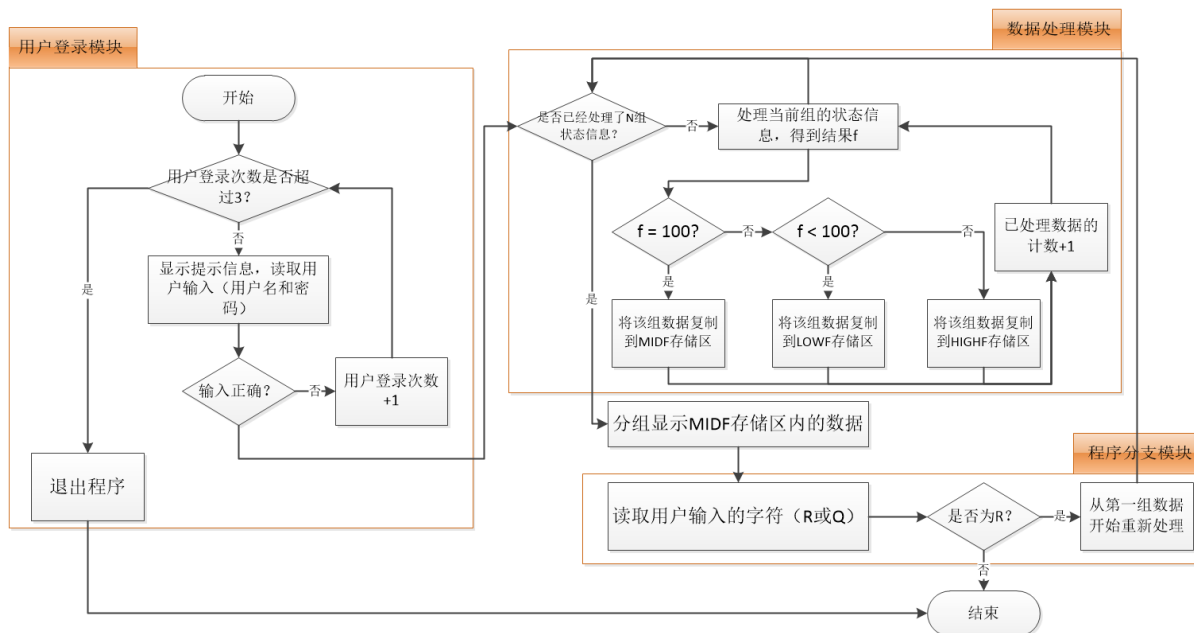


图 3.1 计算机系统运行状态监测系统的程序流程图

3.2.3 实验步骤说明

1. 准备上机实验环境。
2. 使用 Visual Studio 2022 录入源程序, 存盘文件名为 Exp-3.asm。使用 VS 2022 编译并执行源文件, 观察提示信息, 若出错, 则修改错误并存盘后重新编译执行, 直至不再报错为止。
3. 将汇编源代码分解到两个不同的源文件中, 重复以上操作, 直至程序能在 VS 2022 中成功编译并执行。
3. 测试程序基本功能。首先, 连续输入错误的用户名和密码 3 次, 观察现象; 然后先输入 1 次错误的用户名和密码, 再输入一次正确的用户名和密码, 观察现象。在此之后, 在程序窗口中输入 R 并观察现象 (可以多次输入 R 测试程序稳定性); 最后在窗口中输入 Q, 观察现象。
4. 利用反汇编, 重新单步调试执行程序, 观察主程序在调用子程序时堆栈的变化 (栈顶、

汇编语言程序设计实验报告

EBP/ESP 等关键点的变化)，以及执行 CALL 和 RET 指令时 CPU 完成的操作。

5. 再次重新单步调试执行程序，在调用某一子程序且要返回时（RET 指令之前），改变当前的栈顶元素的值，观察执行 RET 指令后 EIP 的变化以及程序的返回点。

6. 在反汇编窗口中观察 invoke 伪指令对应的汇编语句、子程序中局部变量的存储空间对应的地址，以及访问局部变量时地址表达式的特点。

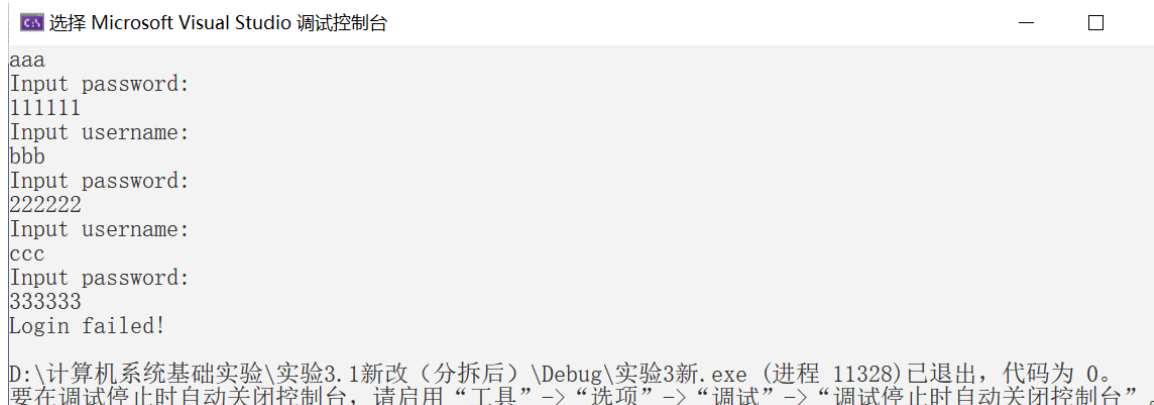
7. 观察模块间参数的传递方法（包括公共符号的定义和外部符号的引用），并观察若符号名不一致或类型不一致时发生的现象。

3.2.4 实验记录与分析

1. 实验环境条件：P3 1GHz，256M 内存；WINDOWS10 下 DOSBox0.73；EDIT.EXE 2.0；MASM.EXE 6.0；LINK.EXE 5.2；TD.EXE 5.0。

2. 在 VS 2022 中依次编译未拆分的汇编源代码和拆分后的汇编源代码文件，二者均能正常通过编译并运行。

3. 运行程序，首先进入用户登录环节，程序显示提示信息“Input username:”和“Input password:”。首先，连续 3 次输入错误的用户名和密码，程序显示 “Login failed!” 并直接退出，如图 3.2 所示。



```
选择 Microsoft Visual Studio 调试控制台
aaa
Input password:
111111
Input username:
bbb
Input password:
222222
Input username:
ccc
Input password:
333333
Login failed!
D:\计算机系统基础实验\实验3.1新改（分拆后）\Debug\实验3新.exe (进程 11328) 已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
```

图 3.2 用户登录失败

然后再次输入一次错误的用户名和密码和一次正确的用户名和密码，程序显示 “Login succeeded!” 并分组显示了 MIDF 存储区中的各组数据（格式为“流水号，数据 a，数据 b，数据 c”），如图 3.3 所示，这说明程序可以正常完成用户登录操作。此时程序提示用户要输入一个字符（R 或 Q）以继续操作。



```
选择 D:\计算机系统基础实验\实验3.1新改（分拆后）\Debug
Input username:
aaa
Input password:
111111
Input username:
ljp
Input password:
123456
Login succeeded!
MIDF存储区中的各组数据:
1, 2540, 1, 1
2, 2540, 2, 2
3, 2540, 3, 3
--
请选择操作（按R键重新执行，按Q键退出）:
```

汇编语言程序设计实验报告

图 3.3 用户登录成功，并显示 MIDF 存储区中的数据

接下来多次输入 ‘R’，每次输入后程序都会重新对各组数据进行处理并输出 MIDF 存储区的数据，且每次执行的结果一致。如图 3.4 所示。

```
选择 D:\计算机系统基础实验\实验3.1新改（分拆后）\I
123456
Login succeeded!
MIDF存储区中的各组数据:
1, 2540, 1, 1
2, 2540, 2, 2
3, 2540, 3, 3
---
请选择操作（按R键重新执行，按Q键退出）:
R
MIDF存储区中的各组数据:
1, 2540, 1, 1
2, 2540, 2, 2
3, 2540, 3, 3
---
请选择操作（按R键重新执行，按Q键退出）:
R
MIDF存储区中的各组数据:
1, 2540, 1, 1
2, 2540, 2, 2
3, 2540, 3, 3
---
请选择操作（按R键重新执行，按Q键退出）:
R
MIDF存储区中的各组数据:
1, 2540, 1, 1
2, 2540, 2, 2
3, 2540, 3, 3
---
请选择操作（按R键重新执行，按Q键退出）:
```

图 3.4 每次输入 R 都会重复执行程序对数据的处理流程

最后输入 ‘Q’，程序退出，如图 3.5 所示。

```
请选择操作（按R键重新执行，按Q键退出）:
Q
D:\计算机系统基础实验\实验3.1新改（分拆后）\Debug\实验3新.exe（进
程 19860）已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”-
>“调试停止时自动关闭控制台”。
按任意键关闭此窗口...
```

图 3.5 输入 Q，程序退出

综上所述，程序完整实现了任务书中要求的功能。

4. （1）重新执行程序并打开反汇编，单步调试程序。以调用子程序 login 为例，在执行指令 “call login”之前，观察堆栈段的情况，如图 3.6 所示。可以看到此时的栈基址指针 EBP=00CFF8CC、栈顶指针 ESP=00CFF888、EIP=000B872A，二者之间的内容就是当前栈的内容，主要存放了主函数中定义的各类变量。

（2）接下来单步执行 “call login” 后，如图 3.7 所示，可以看到 EIP 的值变为 000B8470，其值为当前正在执行指令（即 login 子程序的第一句指令）的地址；ESP 的值变为 00CFF884，相比上一步的 ESP 减少了 4，其原因在于程序调用 CALL 指令时，CPU 会将返回点（即 “call login” 指令的下一条指令）的地址 000B872F 压栈。故此时堆栈栈顶（00CFF884）处的内容是 000B872F。由

汇编语言程序设计实验报告

于 login 子程序没有参数，故没有在此之前依次将参数压栈；反之，若某子程序有参数，则会在将返回点压栈前依次将参数压栈。

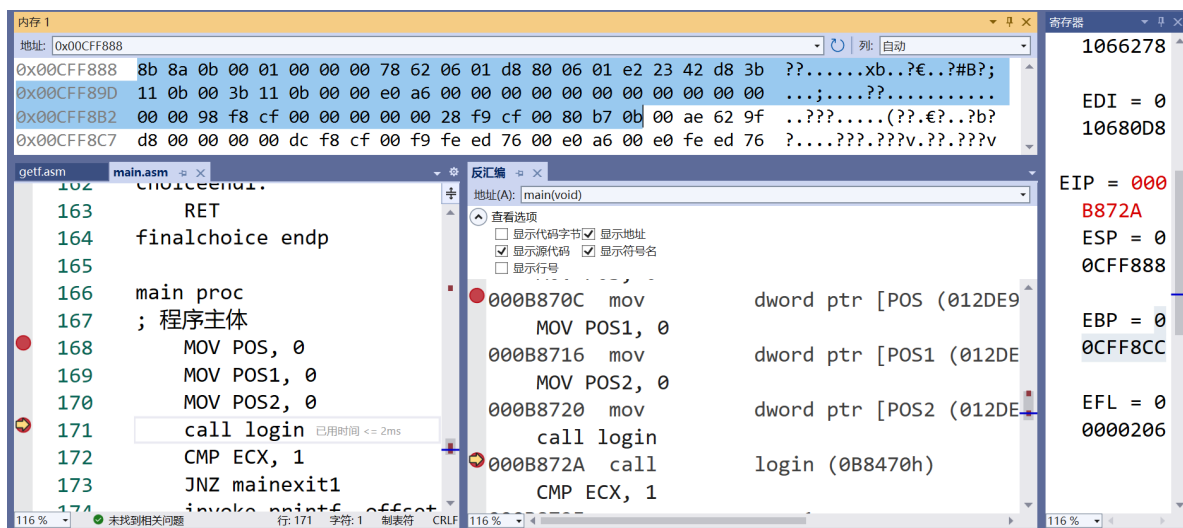


图 3.6 调用 login 子程序前的堆栈状态

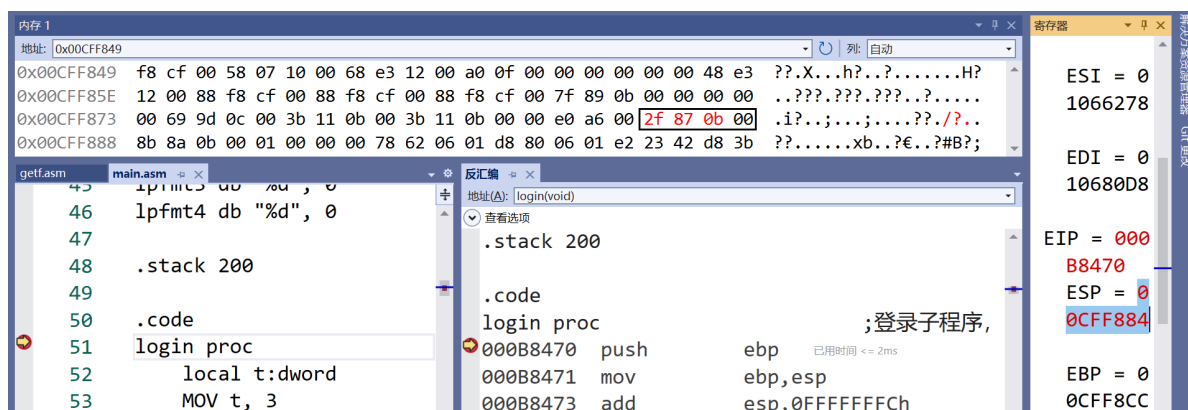


图 3.7 调用 login 子程序时堆栈状态的变化

(3) 观察 login 子程序内部，可以看到在真正进入函数主要功能前，程序首先要执行三条指令：“push ebp” “mov ebp, esp” 和 “add esp, 0FFFFFFFCh”。通常来说，任意一个子程序在从 call 指令进入时首先都会执行前两条指令，其目的在于将原先的 EBP 压栈，并将接下来堆栈的基址 EBP 设为当前的栈顶位置 ESP，这样可以实现对 EBP 的现场保护。而如果子程序定义了局部变量，则在前两条指令后还会将 ESP 的位置减小（上移）。此处由于局部变量 t 的存在，ESP 需要减去 4，恰好是 t 所占的空间大小。

(4) 将断点打在子程序的 RET 指令前，执行到此处后继续执行，观察堆栈状态的变化。从反汇编窗口中可以看出，RET 指令被解释为 “leave” 和 “ret” 两条语句，如图 3.8 所示。执行后，ESP 变为 00CFF884，EBP 变为 00CFF8CC，二者分别与进入 login 子程序前的值相等，说明 leave 指令本质上就是 “mov esp, ebp” 和 “pop ebp”，是 (3) 中前两条指令的逆过程。此后执行 ret 指令时，CPU 会弹出栈顶的返回点并赋给 EIP，程序继续从调用子程序指令的下一条指令开始顺序执行。此时 ESP 再加 4 变为 00CFF888，与 “call login” 指令前的状态完全一致，如图 3.9 所示。因此子程序可以完成保护与恢复现场的工作。

汇编语言程序设计实验报告

(5) 若将栈顶的数值修改为返回点之外的某个值, 则返回时 EIP 的值就是栈顶这个被修改的值, 会从这个地址处的指令开始继续执行。利用此原理可以实现缓冲区溢出攻击。

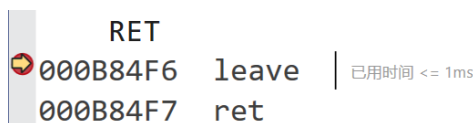


图 3.8 RET 指令的内部解释

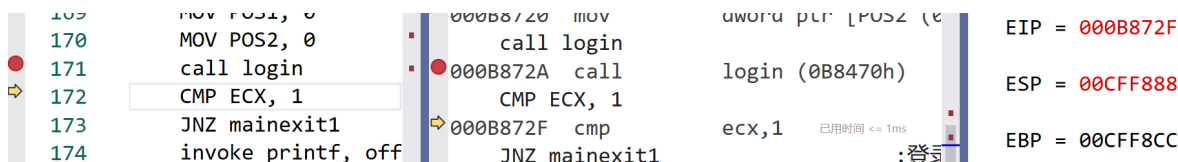


图 3.9 跳出 login 子程序时 EIP、ESP、EBP 寄存器值的变化

5. 观察 invoke 伪指令对应的汇编语句。以 login 子程序中的“invoke printf, offset HINT”和“invoke scanf, offset lpfmt1, offset USERNAMEREAD”指令为例, 其反汇编指令如图 3.10 所示。可以发现 invoke 伪指令本质上是调用子程序来实现相应功能, 在指令中同时还传入了参数。具体实现上, 编译器先将参数入栈, 然后通过 CALL 指令调用子程序, 最后子程序返回后将栈顶指针下移, 相当于释放了此前参数的存储空间。

```
invoke printf, offset HINT
000B8482 push      offset HINT (012DEA2h)
000B8487 call      _printf (0B1BEAh)
000B848C add       esp,4
invoke scanf, offset lpfmt1, offset USERNAMEREAD
000B848F push      offset USERNAMEREAD (012DEEDh)
000B8494 push      offset lpfmt1 (012DF5Eh)
000B8499 call      _scanf (0B11C2h)
000B849E add       esp,8
```

图 3.10 invoke 伪指令对应的反汇编指令及其意义

根据上述分析可知, 子程序中局部变量的存储空间对应的地址位于刚进入子程序时栈基址指针 EBP 的上方, 例如 login 子程序中局部变量 t (占 4 个字节) 的地址表达式为[EBP-4], 根据 EBP=00CFF884 可计算出 t 的地址 00CFF880。访问局部变量时, 地址表达式的特点是“类型名 ptr [变量名]”, 如图 3.11 所示。

```
DEC t
000B84E6 dec      dword ptr [t]
CMP t, 0
000B84E9 cmp      dword ptr [t],0
```

图 3.11 访问局部变量时的地址表达式

6. 进一步观察模块间参数的传递方法 (包括公共符号的定义和外部符号的引用)。公共符号的定义主要是在 .data 节后, 定义格式是“符号名 (变量名) 类型名 初始值”。在不同模块之间, 若要引用外部参数, 需要在文件开头使用 externdef 指令, 格式是“externdef 变量名: 类型名”, 如图 3.12 所示。

若符号名不一致或类型不一致, 则编译无法通过。同名符号类型不一致时, 会报错为“错误

汇编语言程序设计实验报告

A2022 instruction operands must be the same size”，原因是程序在链接时符号解析发生错误，如图 3.13 所示。

```
main.asm  getf.asm  -  X
1  .686P
2  .model flat,c
3  externdef SF:dword, SDA:dword, SDB:dword, SDC:dword
4
5  .code
6  getf proc count: dword          ;获取f的子程序，有入口参数
```

图 3.12 引用外部符号的方法

✖ A2022 instruction operands must be the same size

图 3.13 符号名或类型不一致时会报错

符号名不一致时，汇编器无法找到指令所指的符号，会报错“错误 A2006 undefined symbol: SF”，如图 3.14 所示。

✖ A2006 undefined symbol: SF

图 3.14 符号名或类型不一致时会报错

3.3 任务 3.2 实验过程

3.3.1 实验方法说明

1. 准备上机实验环境。
2. 将任务 3.1 中的汇编代码进行改写，将功能（1）（3）（4）用 C 语言程序实现。
3. 用 C 语言实现“修改采集的状态数据”的功能，即在功能（4）时按 M 键可以输入一组采集数据（不包括 f 字段），用来覆盖 N 组采集到的状态信息中的第一组数据，然后回到功能（4）的等待按键状态。

3.3.2 实验记录与分析

1. 实验环境条件：P3 1GHz，256M 内存；WINDOWS 10 下 Visual Studio 2022。
2. 将相关功能用 C 语言实现，用 VS 2022 编译运行，观察现象。成功登录后，程序首先显示 MIDF 存储区中的各组数据，并提示“请选择操作（按 R 键重新执行，按 Q 键退出，按 M 键输入一组采集数据（不包括 f 字段）并覆盖状态信息中的第一组数据）”。
接着输入 M，新输入一组采集数据“123 3333 5 3970”（满足 $f=100$ ），然后输入 R 重新执行程序。程序第二次输出的 MIDF 存储区中的各组数据中就包含这一组新输入的数据，如图 3.15 所示。
再次输入 M，新输入一组采集数据“200 700 60 5”（满足 $f<100$ ），然后输入 R 重新执行程序。程序第三次输出的 MIDF 存储区中的各组数据中不包含这一组新输入的数据，如图 3.16 所示。
最后输入 Q，程序退出。
3. 仔细观察 MIDF 存储区的变化情况，第一遍执行完成后，SDA、SDB、SDC 存储区的情况如图 3.17 所示。以输入数据“123 3333 5 3970”为例，程序重新执行时，可以看到 SDA、SDB、SDC 存储区前 4 字节的内容都被新输入的数据覆盖，如图 3.18 所示。

汇编语言程序设计实验报告

```
选择 D:\计算机系统基础实验\实验3.2\Debug\实验3.2.exe
Input username:
ljp
Input password:
123456
Login succeeded!
MIDF存储区中的各组数据:
1, 2540, 1, 1
2, 2540, 2, 2
3, 2540, 3, 3
---
请选择操作（按R键重新执行，按Q键退出，按M键输入一组采集数据
（不包括f字段）并覆盖状态信息中的第一组数据）：
M
请输入序列号、状态信息a、b、c，用空格隔开:
123 3333 5 3970
请选择操作（按R键重新执行，按Q键退出，按M键输入一组采集数据
（不包括f字段）并覆盖状态信息中的第一组数据）：
R
MIDF存储区中的各组数据:
123, 3333, 5, 3970
2, 2540, 2, 2
3, 2540, 3, 3
---
```

图 3.15 输入 f=100 的数据并覆盖第一组数据

```
请选择操作（按R键重新执行，按Q键退出，按M键输入一组采集数据
（不包括f字段）并覆盖状态信息中的第一组数据）：
M
请输入序列号、状态信息a、b、c，用空格隔开:
200 700 60 5
请选择操作（按R键重新执行，按Q键退出，按M键输入一组采集数据
（不包括f字段）并覆盖状态信息中的第一组数据）：
R
MIDF存储区中的各组数据:
2, 2540, 2, 2
3, 2540, 3, 3
---
```

图 3.16 输入 f<100 的数据并覆盖第一组数据

内存 1	
地址: 0x0048A00A	
0x0048A00A	ec 09 00 00 ec 09 00 00 ec 09 00 00 04 00 00 00 00 00
0x0048A01D	00 ff ff ff ff fe ff ff ff fd ff ff ff fc ff ff ff 00 00
0x0048A030	00 00 01 00 00 00 02 00 00 00 03 00 00 00 04 00 00 00 00
0x0048A043	00 00 00 ff ff ff ff fe ff ff ff fd ff ff ff fc ff ff ff
0x0048A056	00 00 00 00 01 00 00 00 02 00 00 00 03 00 00 00 04 00 00
0x0048A069	00 00 00 00 00 ff ff ff ff fe ff ff ff fd ff ff ff fc ff
0x0048A07C	ff ff 00 00 00 00 00 00 00 00 04 04 00 00 00 04 00 00 00
0x0048A08F	04 00 00 00 05 00 00 00 00 00 00 00 00 00 00 00 06 ff
0x0048A0A2	ff ff ff ff ff ff ff ff ff ff ff ff ff 07 fe ff ff ff fe ff
0x0048A0B5	ff fe ff ff ff 08 fd ff ff ff fd ff ff ff fd ff ff ff 09
0x0048A0C8	fc ff ff ff fc ff ff ff fc ff ff ff 0a 00 00 00 00 00 00

图 3.17 SDA、SDB、SDC 存储区初始状态

内存 1	
地址: 0x0048A00A	
0x0048A00A	05 0d 00 00 ec 09 00 00 ec 09 00 00 04 00 00 00 00 00
0x0048A01D	00 ff ff ff ff fe ff ff ff fd ff ff ff fc ff ff ff 00 00
0x0048A030	00 00 05 00 00 00 02 00 00 00 03 00 00 00 04 00 00 00 00
0x0048A043	00 00 00 ff ff ff ff fe ff ff ff fd ff ff ff fc ff ff ff
0x0048A056	00 00 00 00 82 0f 00 00 02 00 00 00 03 00 00 00 04 00 00
0x0048A069	00 00 00 00 00 ff ff ff ff fe ff ff ff fd ff ff ff fc ff
0x0048A07C	ff ff 00 00 00 00 00 00 00 00 04 04 00 00 00 04 00 00 00

图 3.18 输入一组数据后 SDA、SDB、SDC 存储区状态

汇编语言程序设计实验报告

4. 根据以上观察，对任务书中的问题进行解答。

(1) C 语言程序中，利用 `extern` 说明外部变量和函数。汇编指令在访问 C 语言程序的变量时，全局变量翻译为[变量名]，局部变量和参数通过 EBP、ESP 访问（如 `dword ptr [ebp-54h]` 等）；而 C 语言语句访问汇编语言定义的变量时，直接翻译为“类型名 [变量名]”，如图 3.19 所示。

(2) 要保证能在 C 语言程序和汇编语言程序中正确访问采集到的状态信息结构中的数据，需要搞清楚三个存储区的首地址、各组数据包含的数据类型（DD）以及所占的空间（4 字节）、以及每组数据在这三个存储区存放的地址。同时还要选对寻址方式，如寄存器间接寻址、变址寻址、基址加变址寻址都是可行的方法。

```
POS = 0;
00481D0A mov     dword ptr [POS (048AE96h)],0
POS1 = 0;
00481D14 mov     dword ptr [POS1 (048AE9Ah)],0
POS2 = 0;
00481D1E mov     dword ptr [POS2 (048AE9Eh)],0
```

图 3.19 C 语言语句访问汇编语言定义的变量

```
int log = login(), choice = 1;
00481CEB call    _login (04811A4h)
```

图 3.20 C 语言函数调用语句对应的汇编语句

(3) 根据不同变量地址之间的关系，可以实现一个变量名称不出现在语句中的情况下也能修改该变量值的功能。这主要通过指针完成，例如，若 `x`、`y` 均为 `int` 型数组变量中的元素，下标分别为 5、10，则可以通过指令“`*(&x+5)=100`”修改 `y` 的值。实际上，（地址）类型转换的含义是将某个地址强行看做是某种类型变量的首地址，并在此基础上对该类型变量进行操作。

例如：“`char a[10]; *(int *)a=123;`”就是把以 `a` 为首地址的 `int` 类型变量（占 4 个字节）改为 123。而反过来，“`int a[10]; *(char *)a=123;`”代表把以 `a` 为首地址的 `char` 类型变量（占 1 个字节）改为 123 对应的字符。

(4) C 语言函数调用语句对应的汇编语句是“`call _汇编语言的函数名`”，如图 3.20 所示。调用函数与被调用函数之间传递信息的方式有多种，包括用寄存器传递入口参数或出口参数、定义内存操作数、采用堆栈方式获取信息等。如果汇编语言子程序的语言风格不是 C 语言风格，则被 C 语言程序调用时也不会出错，因为本质上还是机器指令。

(5) 若在 C 语言程序中不合理地嵌入汇编语言的指令语句，比如在连续几条 C 语言语句中间（尤其是在几行计算公式对应的 C 语言语句中间）加入一条修改 EAX 寄存器（或 EBP、DS 等寄存器）的汇编指令语句，则可能会导致结果完全错误；若在 C 语言函数返回前加入一条修改 EBP 或 ESP 的语句，则会使返回后堆栈状态混乱，甚至可能造成访问越界。

3.4 小结

本次实验中，我学会了汇编语言与 C 语言的混合编程，并用 C 语言编写主函数，调用此前写好的汇编语言程序段完成全部功能。通过对二者的结合使用，我进一步认识到模块化编程在实际工程中的广泛应用，了解了不同模块之间信息传递与组装的基本方法，熟悉了 C 语言中 `invoke` 伪指令的用法及其汇编语言解释，并懂得了 C 语言和汇编语言之间存在相互调用的关系。同时，通过实验中的深入观察，我也掌握了在子程序调用过程中堆栈、EBP、ESP 等关键地方的变化及其原理，还包括对一些变量访问的细节，是对理论知识的一次实证分析。

综上所述，我圆满完成了本次实验任务，对“模块化”一词也有了全新的认识。

4 二进制炸弹破解

4.1 实验内容

本次实验的主要目的与要求有：

- (1) 熟悉动态与静态反汇编工具；
- (2) 熟悉程序的机器级表示，掌握逆向工程的原理与技能；
- (3) 完成执行程序的调试，提升对计算机系统的理解与分析能力。

4.2 任务 4.1 实验过程

4.2.1 实验方法说明

1. 安装 IDA Pro 静态反汇编工具，学习软件用法。在 IDA Pro 中打开二进制炸弹 bomb.exe，观察其中反汇编语句，推断每一关的密码。

2. 在 cmd 中运行 bomb.exe，并测试每一关的密码是否正确。

4.2.2 实验记录与分析

(1) 在 IDA Pro 中打开二进制炸弹 bomb.exe，首先输入“bomb 115652（学号后 6 位）”，观察现象。

(2) 第一关密码的破解

第一关主要考察字符串比较。首先分析炸弹爆炸的流程，发现在输入一个字符串后程序根据输入随机生成密码字符串，然后比对其是否与输入的字符串相同，若相同则进入下一关，否则炸弹爆炸。跟随程序执行步骤可找出答案字符串名为 thestring，进一步查看其内容为连续的 10 个字节“YHQVBHgTnZ”，此即为第一关的密码，如图 4.1 所示。输入后通过第一关，如图 4.2 所示。

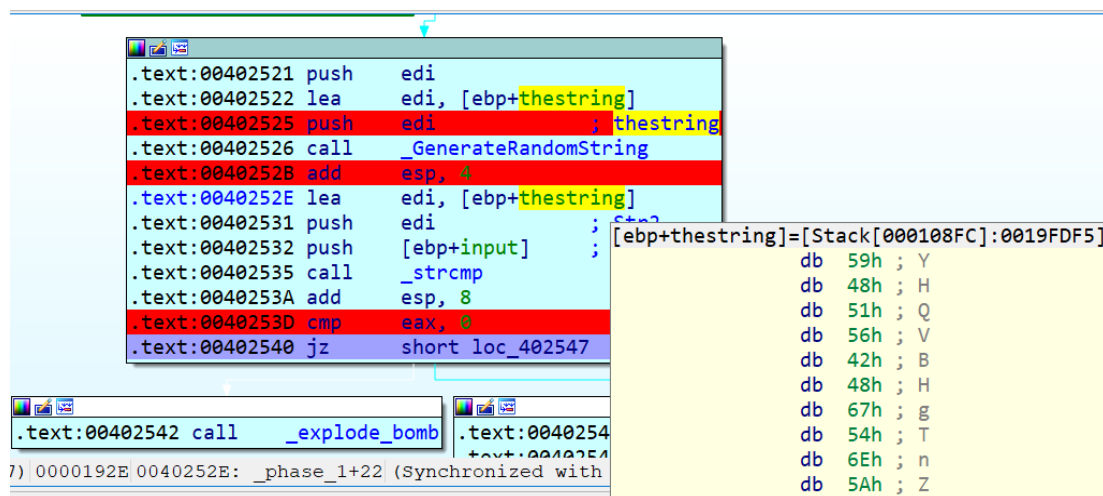


图 4.1 第一关密码的破解

汇编语言程序设计实验报告

超级二进制炸弹，欢迎你！
=====

请输入第1级的密码：YHQVBHgTnZ
牛刀小试`你已经通过了第1级考验！

图 4.2 通过第一关

(3) 第二关密码的破解

第二关考察循环结构。根据输入，程序会自动从 phase_2_0 至 phase_2_9 中选择一个执行，此处程序进入了 phase_2_3，如图 4.3 所示。

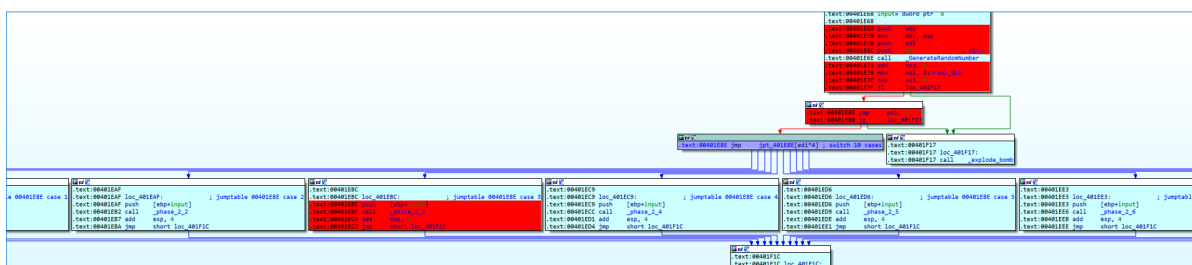


图 4.3 程序根据输入选择 10 个流程中的一个

根据程序单步执行到最后的判断模块，可以发现此处必须输入数字，且位数只能为 6 (numbers 共占 10 个字节，存储了输入数字的前半部分)，如图 4.4 所示。接下来可以看到程序分 3 次比较数，比较的对象是第 i 个数和第 6-i 个数，若二者相同则继续，否则炸弹爆炸，如图 4.5 所示。因此形如 “1 2 3 3 2 1” 或 “1 3 5 5 3 1” 等的密码都是正确的，输入后通过第二关，如图 4.6 所示。

```
[ebp+numbers]=[Stack[00010D6C]:0019FDD4]
db 1
db 0
db 0
db 0
db 2
db 0
db 0
db 0
db 3
db 0
```

图 4.4 第二关输入要求只能为 6 位数字串

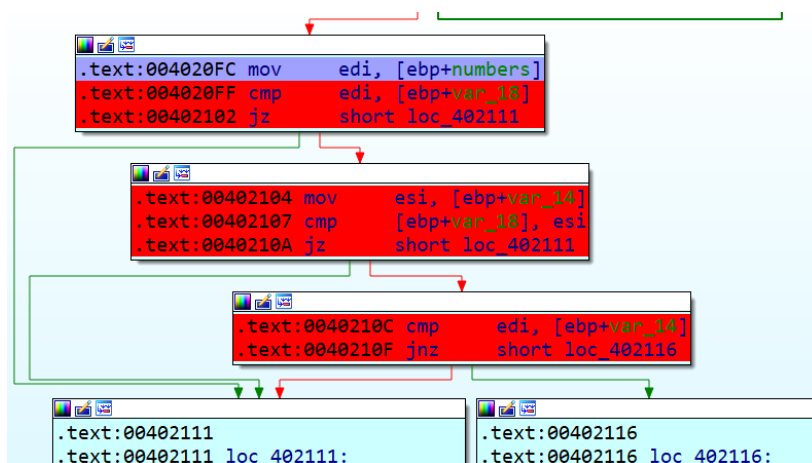


图 4.5 三次循环比较

汇编语言程序设计实验报告

请输入第2级的密码: 1 2 3 3 2 1
不错不错~你已经通过了第2级考验!

图 4.6 通过第二关

(4) 第三关密码的破解

第三关考察条件/分支结构。根据输入，程序同样会自动从 phase_3_0 至 phase_3_2 中选择一个执行，此处程序进入了 phase_3_0。此后，程序首先比较输入的第一个数与 7 的大小，若第一个数大于 7 则炸弹爆炸。故密码的第一位只能是 0~7 这 8 个数字中的一个。在该模块内部还有一次分支，分支会从 8 个小模块中再次选择一个，这每个模块正好对应第一位数，如图 4.7 所示。

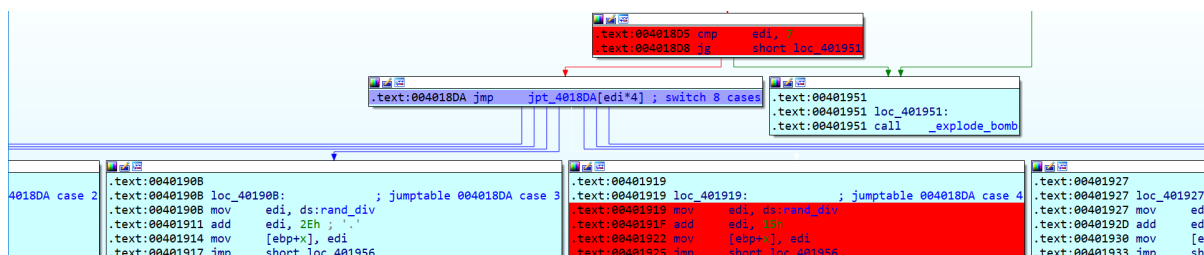


图 4.7 第三关的分支结构（根据输入的第一位数决定）

接下来逐步分析程序的操作。进入分支后，程序首先比较输入的第二位数是否为 73h（即 115），如图 4.8 所示。

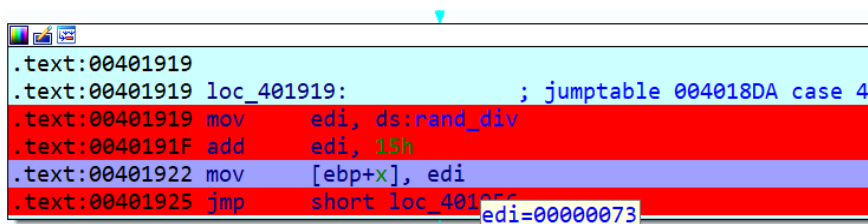


图 4.8 进入分支后判断第二位输入是否正确

但是程序并没有对第三位输入数进行检验（甚至只输入前两位数也能通过）。因此形如“4 115 (...)”的密码都可以通过第三关，如图 4.9 所示。

请输入第3级的密码: 4 115 3
今夜没加班? 你已经通过了第3级考验!

图 4.9 通过第三关

4.3 小结

通过本次实验，我学习了 IDA Pro 静态反汇编工具的用法。通过对二进制炸弹的破解，我进一步掌握了通过反汇编语句来解释源程序逻辑的方法，锻炼了我逆向工程处理问题的能力。虽然二进制炸弹的 4 关中我只通过了 3 关，但我仍然从中收获了很多，包括对程序中常见结构（分支、循环等）的判断、源程序与反汇编关系的认识等。相信本次实验也会为我在将来处理问题时提供一种全新的思路。

5 不同指令集体系结构和编程环境的比较

5.1 实验内容

本次实验的主要目的与要求有：

- (1) 了解 ARM + Linux 环境下程序设计的特点及配套的开发工具；
- (2) 观察并理解 80X86+windows 和 ARM+Linux 下，不同的“指令集体系结构+编程环境”的基本特点。

5.2 任务 5.1 实验过程

5.2.1 实验方法说明

1. 准备上机实验环境，安装 QEMU 等环境，熟悉其用法。
2. 编译执行“ARM 虚拟环境安装说明”文档中 1.4.1 的程序（一个显示 Hello World 的汇编语言程序）。
3. 编译执行“ARM 虚拟环境安装说明”文档中 2.2.1（一个测试内存拷贝函数的执行时间的 C 语言与汇编语言混合编程的程序）和 2.2.3（对前面 2.2.1 程序的优化）的程序。

5.2.2 实验记录与分析

1. 实验环境条件：P3 1GHz，256M 内存；WINDOWS 10 下 QEMU 虚拟机环境。
2. 在 qemutest 文件夹下以管理员身份运行 cmd，进入到 D:\qemutest 目录下，打开 QEMU 终端，选择 view-serial0 选项后登录。用 C 语言编写 hello world 测试程序，输入“vi hello.s”，录入源程序后按 ESC 键并输入“.wq”保存。接着输入“as hello.s -o hello.o”“ld hello.o -o hello”生成二进制目标文件和可执行文件，最后输入“./hello”运行，如图 5.1 所示。
3. （1）用同样的方法编写测试内存拷贝时间的 C 语言程序 time.c 和 copy.s，如图 5.2 所示。执行命令“gcc time.c copy.s -o m1”编译运行，可以看到 memorycopy 函数具体的执行时间为 46924066ns，如图 5.3 所示。接下来对 copy.s 的代码进行优化。
（2）考虑内存突发传输方式的优化。由于此前每次读写都是以字节为单位进行的，效率较低，故可以考虑一次性对多个字节进行读写。于是将 copy.s 改为一次读写 16 个字节，并将改进后的代码保存至 copy21.s。重复上述操作编译运行，可以看到 memorycopy 函数的执行时间为 13087715ns，如图 5.4 所示。显然一次性读写 16 个字节的效率明显高于单字节读写。

汇编语言程序设计实验报告

5.2.3 查阅资料并比较 ARMv8 与 80X86 体系的异同

上述实验是基于华为鲲鹏服务器所采用的 CPU（即 ARMv8 系列）的汇编语言编程完成。与 80X86 体系下的汇编语言相比，二者在 CPU 内寄存器、段的定义方法、指令语句及格式的特点、子程序调用的参数传递与返回方法、与 C 语言混合编程、开发环境等方面均有所不同，下面分别进行比较。

（1）CPU 内寄存器

80X86 的 CPU 内寄存器包括了数据寄存器（EAX、EBX、ECX 和 EDX）。其中 EAX 是累加器；EBX 为基址寄存器，用于存储内存中数据存放的（栈的）基址；ECX 为计数寄存器，在循环和字符串操作中用 ECX 来控制循环次数，而在位操作中可用 CL 指明移位的位数；EDX 为数据寄存器，在乘/除运算中可作为默认操作数参与运算，也可用 EDX 存放 I/O 端口地址。除此之外，80X86 的 CPU 内寄存器还包括指示器变址寄存器（ESI、EDI、ESP、EBP）、指令指针寄存器（EIP）、段寄存器（CS、SS、DS、ES、FS、GS）等。

而 ARMv8 的 CPU 内寄存器主要包括通用寄存器（数据寄存器）和指针寄存器。通用寄存器由 31 个 64 位的寄存器组成，编号为 X0~X30，一般用于数据储存，处理速度更快、效率更高。指针寄存器则显得比较复杂，如表 5.1 所示。

表 5.1 ARMv8 下的指令寄存器种类

分类	名称	功能	属性
异常寄存器	far	保存发生异常的虚拟地址	32 位
	esr (csr)	保存操作系统内核的触发异常原因	32 位
	exception	保存中断事件请求信号	32 位
浮点寄存器	v	向量寄存器，可通过指令访问特定位数	32 个 128 位寄存器，编号为 v0~v31
	s	单精度寄存器	32 个 32 位寄存器，编号为 s0~s31
	d	双精度寄存器	32 个 64 位寄存器，编号为 d0~d31
通用寄存器	fp	栈帧指针寄存器，一般情况下是通用的，有时会利用 fp 存储栈的基址	16 位或 32 位
	sp	栈指针寄存器，保存栈顶地址，类似于 esp	16 位或 32 位
	pc	指令指针寄存器，保存下一次要执行的指令地址	16 位
.....			

可见两种体系在 CPU 内寄存器上有较大差异。

（2）段的定义方法

在 80X86 下，段的定义中.model、.data、.stack 和.code 四种较常用，分别代表函数原型定义、数据段定义、堆栈段定义和代码段定义。而在 ARMv8 中，代码部分用.text 来定义，全局函数的定

汇编语言程序设计实验报告

义使用.global 函数名定义。

(3) 指令语句及格式的特点

在指令集上，80X86 和 ARMv8 的区别是前者使用复杂指令集（CISC），能通过单个指令执行复杂的操作。而后者使用精简指令集（RISC），具有更少数量的通用指令，功率效率更高。在任务 5.2 中，汇编代码使用了 ldrb（读取数据到寄存器中）、str（从寄存器中取数据传递到存储器）指令，注意格式为“ldrb/str 源寄存器，目的地址，偏移量”，与 80X86 中的变址寻址格式不同。

(4) 子程序调用的参数传递与返回方法

通过 gdb 工具观察 ARMv8 体系下的子程序调用参数传递与返回方法，可以发现函数调用时传参的顺序是从寄存器 r0 到 r3，即第一个参数在 r0 中，第二个参数在 r1 中，以此类推，CPU 在调用子程序时依次从上述寄存器中取数据。而在 80X86 下，根据任务 3.1，可知参数都存在堆栈中，且在进入子程序后参数位于 EBP 的下方，只需通过变址寻址即可取得参数。

(5) 开发环境

80X86 的代码可以在 Windows 操作系统下编译运行，兼容性好；而 ARMv8 下的代码则需要配套 Linux 操作系统进行编译运行，故本实验中采用了基于 Linux 内核的操作系统 openEuler 完成。

5.3 小结

通过本次实验，我了解了 ARM + Linux 环境下程序设计的特点及配套的开发工具（QEMU 虚拟机、openEuler 操作系统），成功地在相应环境下编译运行了任务书所给的程序。通过对程序运行的调试与观察，我进一步了解了 ARMv8 架构下 C 语言与汇编混合编程的方法与技巧，加深了我对一种新的指令集架构的理解。同时，通过对比 80X86+windows 和 ARM+Linux 两种不同指令集体系的异同点，懂得了不同的“指令集体系结构+编程环境”的基本特点，并从原理角度掌握了更多的底层汇编语言知识。

6 中断处理

6.1 实验内容

本次实验的主要目的与要求有：

- (1) 通过观察与验证，理解中断矢量表的概念；
- (2) 熟悉 I/O 访问、BIOS 功能调用方法；
- (3) 掌握实方式下中断处理程序的编制与调试方法；
- (4) 进一步熟悉内存的一些基本操纵技术。

6.2 任务 6.1 实验过程

6.2.1 实验方法说明

1. 准备上机实验环境。
2. 利用中断功能调用的性质编写代码，在 DOSBox 中编译运行，观察是否实现了实时时间显示和驻留效果。
3. 利用 TD 观察中断矢量表和中断处理程序。

6.2.2 设计思想

本题要利用中断实现实时时间显示，且要表现出驻留效果。对于新安装的中断处理程序，需要具有以下功能：

1. 首先执行原中断服务程序的功能（利用堆栈使用 PUSH+CALL 组合，进入中断处理程序后利用 IRET 指令返回原中断服务程序）。
2. 用 I/O 指令从 CMOS 芯片中读取当前时钟的时、分钟和秒信息。
3. 判断本次读取的信息与上次读取的信息是否相同（也可以用计数 18 次来判断是否需要执行显示时分秒的程序），相同则中断返回，不相同则调用显示程序将新的时分秒信息显示到指定位置。
4. 利用 IRET 指令中断返回至原来的 8 号时钟中断程序，并退出。

主程序的处理流程是：

1. 检查中断处理程序是否已经安装，若已经安装则退出。
2. 安装中断处理程序（接管 8 号时钟中断）。
3. 驻留退出程序。

6.2.3 流程图

图 6.1 和图 6.2 是任务 6.1 利用中断实现实时时间显示的主程序和中断处理程序流程图。

汇编语言程序设计实验报告

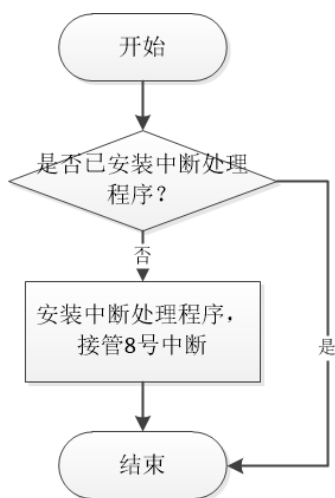


图 6.1 利用中断实现实时时间显示的主程序流程图

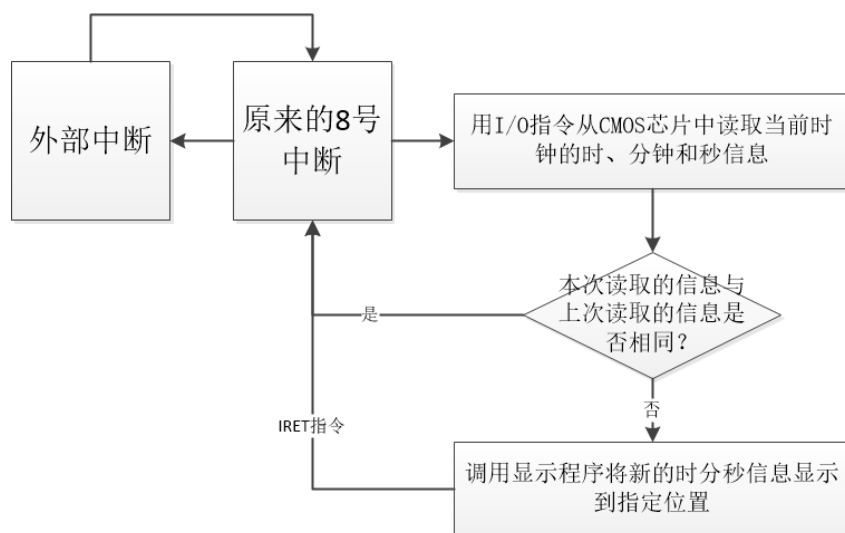


图 6.2 利用中断实现实时时间显示的中断处理程序流程图

6.2.4 实验记录与分析

1. 实验环境条件：P3 1GHz，256M 内存；WINDOWS 10 下 DOSBox0.73；TD.EXE 5.0。
2. 按要求编写程序，在 DOSBox 中编译运行。首先执行一次程序，程序在窗口右上角实时显示时间，并完成中断处理程序的安装，如图 6.3 所示。

接下来按 Q 键，程序退出后，窗口右上角仍然能实时显示时间，如图 6.4 所示。

再运行一次主程序，此时由于中断处理程序已安装，故会提示 “Already installed”，且窗口右上角仍然能实时显示时间，如图 6.5 所示。

汇编语言程序设计实验报告

```
Welcome to DOSBox v0.73

For a short introduction for new users type: INTRO
For supported shell commands type: HELP

If you want more speed, try ctrl-F8 and ctrl-F12.
To activate the keymapper ctrl-F1.
For more information read the README file in the DOSBox directory.

HAVE FUN!
The DOSBox Team http://www.dosbox.com

Z:\>SET BLASTER=A220 I7 D1 H5 T6

Z:\>mount c: d:\masm
Drive C is mounted as local directory d:\masm\

Z:\>c:
C:\>6
```

图 6.3 首次执行主程序，安装中断处理程序并显示实时时间

```
Welcome to DOSBox v0.73

For a short introduction for new users type: INTRO
For supported shell commands type: HELP

If you want more speed, try ctrl-F8 and ctrl-F12.
To activate the keymapper ctrl-F1.
For more information read the README file in the DOSBox directory.

HAVE FUN!
The DOSBox Team http://www.dosbox.com

Z:\>SET BLASTER=A220 I7 D1 H5 T6

Z:\>mount c: d:\masm
Drive C is mounted as local directory d:\masm\

Z:\>c:
C:\>6
C:\>_
```

图 6.4 退出程序，仍能显示实时时间

```
Welcome to DOSBox v0.73

For a short introduction for new users type: INTRO
For supported shell commands type: HELP

If you want more speed, try ctrl-F8 and ctrl-F12.
To activate the keymapper ctrl-F1.
For more information read the README file in the DOSBox directory.

HAVE FUN!
The DOSBox Team http://www.dosbox.com

Z:\>SET BLASTER=A220 I7 D1 H5 T6

Z:\>mount c: d:\masm
Drive C is mounted as local directory d:\masm\

Z:\>c:
C:\>6
C:\>6
Already installed
```

图 6.5 再次执行主程序，可以检测到已安装中断处理程序

汇编语言程序设计实验报告

综上，可以看出程序实现了利用中断显示实时时间的功能，且可以驻留。

3. 在 TD 下观察中断矢量和已有的某个中断处理程序的代码。执行到“MOV AH, 3508”和“INT 21H”后，(fs): (bx) 中存放的就是中断号为 8 的服务程序入口地址，即 fs:0020=E00000F1h (应解释为 IP=00E0h, CS=F100h)，如图 6.6 所示。当然，根据实方式下中断向量的特点，中断向量表起始地址为物理地址 0，每个中断号占 4 个字节，8 号中断对应的中断处理程序入口地址就在 0:[0020H]，直接输入“Goto 0:[0020H]”也能得到同样的结果。

cs:00B6 B80000	mov	ax,0000	ax 0000
cs:00B9 BB2000	mov	bx,0020	bx 0020
cs:00BC 8B0E0000	mov	cx,[0000]	
cs:00C0 8B0E2000	mov	cx,[0020]	dx 0000
cs:00C4 1E	push	ds	si 0000
cs:00C5 0B00	or	ax,[bx+si]	di 0000
cs:00C7 8C060D00	mov	[000D],es	bp 0000
cs:00CB BA2300	mov	dx,0023	sp 00C8
cs:00CE B80825	mov	ax,2508	ds 0AA0
cs:00D1 CD21	int	21	es 0AA0
cs:00D3 EB09	jmp	00DE	ss 0AB0
cs:00D5 BA0F00	mov	dx,000F	cs 0ABD
cs:00D8 B409	mov	ah,09	
cs:00DA CD21	int	21	
cs:00DC EB0C	jmp	00EA	

fs:0020 E0 00 00 F1 D6 0C FA 07 α ±n♀·

ss:00D0

图 6.6 获取 8 号中断处理程序的地址（见下方内存区域）

在代码段使用“jmp es:[00E0]”指令前往中断号为 8 的原中断处理程序的代码段，如图 6.7、图 6.8 所示。

cs:00D3 26FF26E000 jmp es:[00E0] ↓

图 6.7 跳转到已存在的 8 号中断处理程序入口地址

cs:38FE 46	inc	si	ax 2508
cs:38FF EE	out	dx,a1	bx 00E0
cs:3900 7402	je	3904	cx 0000
cs:3902 33F6	xor	si,si	dx 0023
cs:3904 0BF6	or	si,si	si 0000
cs:3906 7425	je	392D	di 0000
cs:3908 57	push	di	bp 0000
cs:3909 E8DEF7	call	30EA	sp 00C8
cs:390C 59	pop	cx	ds 0ABD
cs:390D 8BF0	mov	si,ax	es F100
cs:390F 0BC0	or	ax,ax	ss 0AB0
cs:3911 741A	je	392D	cs 0ABD
cs:3913 EB0D	jmp	392D	
cs:3915 57	push	di	
cs:3916 E8D1F7	call	30EA	

图 6.8 观察已存在的 8 号中断处理程序段

4. 访问 CMOS 中某个单元内容的方法：查阅数据地址对照表，使用指令“OUT OPD, OPS”将 OPS（内部寄存器）中的内容送到 OPD（外部硬件）中，使用指令“IN OPD, OPS”将 OPS（外部硬件）中的内容送到 OPS 中（内部寄存器）。本任务中，硬件时间的秒、分钟、小时信息的地址分别为 00H、02H、04H，结合以上指令即可实现对 CMOS 中数据的读取。

汇编语言程序设计实验报告

6.3 小结

通过本次实验,我通过修改中断处理程序实现了实时显示时间的功能,进一步加深了我对中断、中断处理、中断矢量表的理解,以及对 CMOS 芯片中信息的存取能力。到目前为止,可以发现中断和子程序的调用其实有很多相似之处,例如参数的入栈、返回等,只不过中断涉及到外部硬件设备对 CPU 发出的信号,而子程序调用则是软件内部逻辑上的调用。同时,通过在 TD 中对中断向量表进行观察,我也认识到实方式下中断向量号 i 对应的中断处理程序入口地址就是物理地址 $4*i$,这一特点有助于在编程时迅速找到中断处理程序并作出相应修改。

至此,本学期的计算机系统基础实验就告一段落了。通过这六次实验,我从一个完全没有接触过汇编语言的人变成了一个拥有充分基础理论知识的人。同时,在实践中,我的汇编语言编程、反汇编操作、模块化编程和系统底层认识能力都有了很大的提升,也会根据实际需求用汇编语言写出一些简单的程序。可以说,整个实验课带给我的不仅是代码能力的扎实提高,更是对自己学习能力的磨砺。最后,我要感谢李海波老师在实验中给我的悉心帮助,也要感谢无数个夜晚耐心调 bug 的自己!

汇编语言程序设计实验报告

参考文献

- [1]许向阳. x86 汇编语言程序设计. 武汉：华中科技大学出版社，2020（第 2 章、第 3 章、第 4 章、第 5 章）
- [2]许向阳. 80X86 汇编语言程序设计上机指南. 武汉：华中科技大学出版社， 2007
- [3]王元珍，曹忠升，韩宗芬. 80X86 汇编语言程序设计. 武汉：华中科技大学出版社，2005
- [4]计算机系统基础课程组. 《2022 计算机系统基础实验任务书》，2022
- [5]袁春风. 计算机系统基础（第二版）. 北京：机械工业出版社，2019
- [6]IDA 详细使用指南. <https://blog.csdn.net/oskfh/article/details/127747536>，2022 年 11 月 8 日
- [7]IDA Pro 权威指南. Chris Eagle，石华耀，段桂菊，北京：人民邮电出版社，2012